

Evolution Patterns: Designing and Reusing Architectural Evolution Knowledge to Introduce Architectural Styles

Dalila Tamzalit*, Tom Mens,**

Abstract

Software architectures are critical in the successful development and evolution of software-intensive systems. While formal and automated support for architectural descriptions has been widely addressed, their evolution is equally crucial, but significantly less well-understood and supported. In order to face a recurring evolution need, we introduce the concept of *evolution pattern*. It formalises an architectural evolution through both a set of concepts and a reusable evolution process. We propose it through the recurring need of introducing an architectural style on existing software architectures. We formally describe and analyse the feasibility of architectural evolution patterns, and provide a practical validation by implementing them in **COSA-Builder**, an Eclipse plug-in for the **COSA** architectural description language.

Key words: software evolution, reuse of evolution knowledge, pattern, software architecture, architectural style, graph transformation, architecture description language.

1. Introduction

As acknowledged by Perry and Wolf (1992); Shaw and Garlan (1996a); Bass et al. (1997); Taylor et al. (2009), *software architectures* have become accepted as one of the main artefacts of software development. They form an integral part of the specification of a wide variety of complex software-intensive systems. Including architectural design in the early stages of the

*Université de Nantes, France

**Software Engineering Lab, Université de Mons, Belgium

software system life-cycle can be decisive for the software’s success. They provide a powerful abstraction mechanism that is critical to support the successful development and evolution of the software systems they describe

The growing importance of software architecture descriptions and the maturity of the research field led to the ISO/IEC Standard 42010 (2007) that defines a software architecture as “*the fundamental organization of a system embodied in its components, their relationships [...] and the **principles guiding its design and evolution.***” Research on such principles for guiding architectural evolution has received relatively little attention, despite the promise of controlling cost and other change-related challenges (cf. Le Goaer et al. (2008); Garlan et al. (2009); Mens et al. (2010)).

Our main objective is to offer a disciplined way to reuse evolution knowledge within software architectures. We focus on a specific and recurring evolution need: introducing an architectural style by restructuring an existing software architecture. We propose to formalise and automate *evolution patterns* as a means to guide and support evolution of architectural descriptions. The overall approach follows three successive stages to specify a reusable evolution process: (1) *reify architectural concepts* that may evolve; (2) specify a minimal set of *recurring evolution operations* on these concepts; and (3) specify *the evolution process* through a specific workflow of evolution operations applied to identified architectural concepts.

This article is structured as follows. Section 2 introduces and explains the necessary architectural concepts. Section 3 presents a case study using the COSA architectural description language. Section 4 formally presents and analyses the approach using graph transformations. Section 5 explains how we implemented and validated these ideas in COSABuilder, an Eclipse plug-in for COSA. Section 6 discusses related work, Section 7 highlights some avenues of future research and Section 8 concludes.

2. Architectural concepts

The use of architectural descriptions has become well-established. They specify the various concerns of the system at a high level of abstraction. Such descriptions are made possible thanks to primary notions and concepts of viewpoints and views, architectural description languages and architectural styles.

The architectural description of a software-intensive system is commonly organized in several representations, like the different types of architectural

blueprints in building construction. The objective is to reduce complexity and to facilitate system understanding Clements et al. (2002); Kruchten (1995). We base our work on the ISO/IEC Standard 42010 (2007), partly illustrated in Figure 1 by the shaded dotted rectangle: a description of the *Architecture* of a *System* is composed of *Views* expressed along different *Viewpoints* addressing *Concerns* that are important to a particular set of *Stakeholders*.

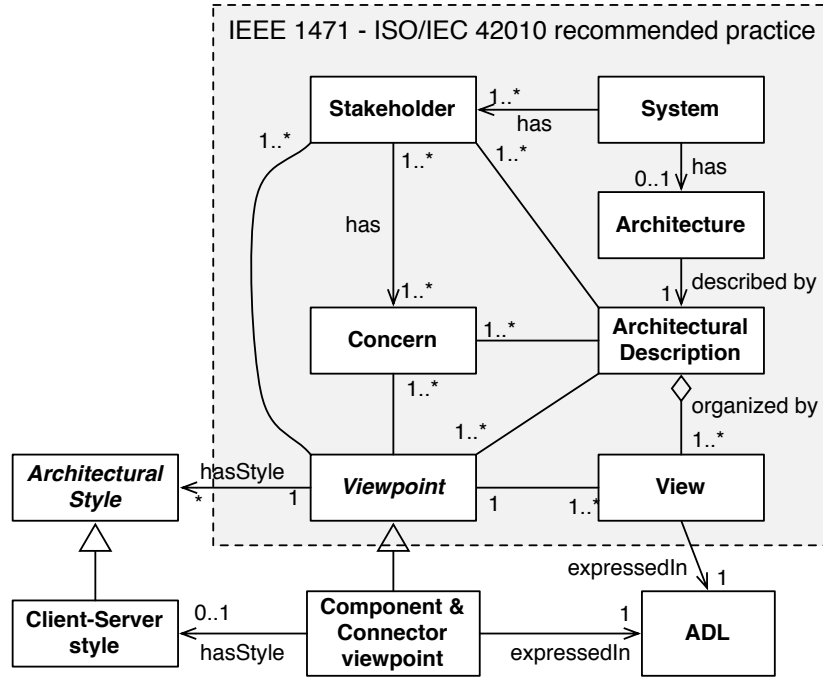


Figure 1: Fragment of the ISO/IEC standard 42010 conceptual framework enriched with the architectural style concept and a particular viewpoint.

Viewpoints are generic and can concern several architectures, while views are architecture-specific. Viewpoints thus allow to organise an architectural description where each view conforms to a particular viewpoint. According to ISO/IEC Standard 42010 (2007), one of the main viewpoints is the *structural* viewpoint. Following Vestal (1993), it structures the organization of an architectural description in terms of coarse-grained *components* with *ports* and their interactions through *connectors* with *roles*, ignoring technical and implementation details. As suggested in Figure 1, the current article focuses on structural viewpoint. We will use from now on the term *architecture* to

refer to *architectural description within the structural viewpoint*.

Architectural Description Languages (ADLs) have been proposed as a formal means to describe software architectures following the structural viewpoint. Many ADLs have been proposed over the years: *ACME* by Garlan et al. (1997), *Aesop* by Garlan et al. (1994), *C2SADEL* by Medvidovic et al. (1999), *Darwin* by Magee et al. (1995), *MetaH* by Vestal and Binns (1993), *Rapide* by Luckham et al. (1995), *SADL* by Moriconi and Riemenschneider (1997), *Unicon* by Shaw et al. (1995), *Wright* by Allen and Garlan (1996) and *AADL* by Lewis (2002). Each of them has its own notation and features, often with their own supporting methods and tools. They generally propose the same concepts of *component*, *port*, *connector* and *role*.

As advised by ISO/IEC Standard 42010 (2007) and Medvidovic and Taylor (2000), an ADL should also provide support to *evolve* architectural descriptions. However, many ADLs, generally domain-specific ones, do not support such architectural evolution. Those that do, typically rely on mechanisms offered by the underlying programming language, such as subtyping and inheritance or refinement. The architectural concepts that are subject to evolution are typically *components* and *connectors* for those ADLs that support them as first-class entities. Shaw and Garlan (1996b) introduced the concept of *architectural style* as a disciplined mechanism to guide the design and use of architectures. Clements et al. (2002); Garlan and Shaw (1993); Garlan et al. (1994) refer to an *architectural style* as a family of architectures in terms of a *pattern of structural organization* through a coordinated set of architectural constraints. These architectural constraints define: a unified vocabulary of component and connector types; constraints on relations between these types; and a semantic interpretation for each instantiated element. Gomaa and Farrukh (1998) rely on architectural styles to facilitate the *construction* of architectures, while Garlan et al. (1994); Shaw and Garlan (1996b) use styles to constrain and evolve architectures.

Among the best known structural architectural styles are the *Pipe-and-Filter* style of Garlan et al. (2002) and the *Client-Server* style of Clements et al. (2002). Any of these architectural styles defines specific types of components, ports, connectors and roles in addition to a set of architectural constraints. In this article we focus on the *Client-Server* architectural style as a case study. Figure 1 shows how to fit it into the ISO/IEC conceptual framework.

Evolution mechanisms proposed by ADLs are generally tied to supporting tools, and thus hardly reusable when a similar architectural evolution

situation occurs. An explicit specification of *evolution pattern* would enable future reuse, thereby reducing the costs and risks of architectural evolution in the long run. A typical example of such a reoccurring pattern is the restructuring of a monolithic architecture of a legacy system into a distributed client-server architecture.

Evolution patterns can be specified in terms of more elementary predefined architectural *evolution operations*. While an elementary evolution operation can lead a given architecture to an inconsistent state, an evolution pattern allows to evolve an architecture from a consistent state to another consistent one. As an evolution pattern should be reusable, its well-formedness needs to be validated before being proposed for reuse in similar evolution situations on different architectures. Upon application of the evolution pattern, automatic analyses can be performed to determine the conformity of the evolved architecture, and to report or resolve any conformance problems that may arise.

3. Case Study

3.1. The COSA ADL

In this article, we will illustrate our approach through the specification, analysis and execution of an evolution pattern to introduce the Client-Server architectural style on a monolithic e-shop architecture. To do so, we will rely on the COSA ADL developed by Maillard et al. (2007) and its associated tool COSABuilder. We use COSA for four main reasons: *(i)* it is generic and extensible, defined through a metamodel; *(ii)* it manipulates architectural elements (configuration, component, connector, port, role, ...) as first-class entities; *(iii)* it can be easily extended with new first-class concepts, which is very useful if we want to add evolution operations and evolution patterns; *(iv)* the metamodel and source code of COSABuilder are available to us.

The notion of computation (represented by *components*) is separated from the notion of interaction and communication (represented by *connectors*). A component has a set of ports (provided or required). The topological structure of the architecture is represented within a *configuration*, a graph of interconnected components. Each composite component has its own configuration that handles its internal architectural elements. Connectors can either be user-defined or built-in. In the latter case, we distinguish between *attachments* (to connect a port to a role) and *bindings* (to interconnect two ports or two roles, generally in case of delegation).

3.2. The e-shop architecture

Figure 2 presents the monolithic e-shop architecture in COSA. It contains three main components: **Product**, **Customer** and **Order** with their own ports in addition to a set of connectors. We only explain one component, **Customer**, as the two other components follow the same spirit. **Customer** has four required ports (**UserDetails**, **Pwd**, **AcceptBill** and **Pay**) and three provided ports (**Authenticate**, **CreateCustomer** and **Bill**). For aims of clarity, the names of these provided ports are hidden since the connectors they are attached to have the same names.

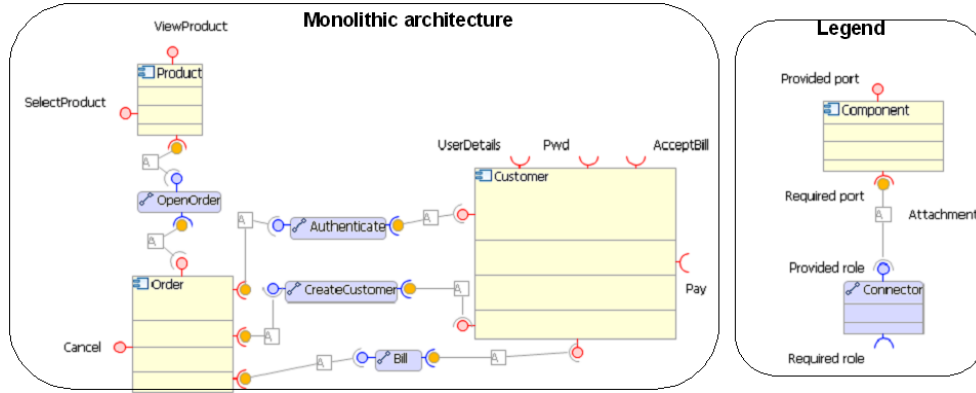


Figure 2: The EShop monolithic architecture expressed using the COSA ADL.

The e-shop architecture also specifies how its components interact together through four connectors: **OpenOrder**, **Authenticate**, **CreateCustomer** and **Bill**. Each of these connectors has two roles, to connect a provided port of one component to a required port of another component. Some component's ports, such as **UserDetails** are connection-free and represent the interaction points of e-shop with its environment.

3.3. Extending COSA with structural dependencies

Since we focus on the structural viewpoint in this article, we restrict ourselves to expressing *structural dependencies*. Ideally, architectural restructuring, expressed by evolution patterns, should preserve dependencies: if some provided port of a component (transitively) depends on a required port of a (possibly different) component, this should remain the case after the restructuring.

In order to address properly architectural restructuring, we need to consider all structural dependencies between ports of components. These dependencies can be of two kinds. *External dependencies* between different components are expressed using connectors, attachments and bindings. *Internal dependencies* between ports of the same component are generally not explicitly materialized. We enriched the COSA metamodel and syntax with a new built-in connector type, named *uses* (*U*), to represent these internal dependencies. The *Product* component in Figure 5(a) has two such internal dependencies: port **SelectProduct** *uses* port **ViewProduct** (a customer first needs to view products in order to select one), and port **OpenOrder** *uses* port **SelectProduct** (a customer can only order selected products). Although not explicitly shown in Figure 2 for aims of readability, the e-shop architecture is enriched with these internal structural dependencies.

3.4. Introducing the Client-Server architectural style

Architectural restructurings, such as the migration to a Client-Server architecture typically need to preserve internal dependencies. Generally used for network-based applications, the Client-Server style proposes two additional types of component, *Server* and *Client*, that are connected together. A software architecture conforming to the Client-Server style is only allowed to have instances of the element types specified by the style. In addition, it must respect all constraints imposed by the style. A server component offers a set of services to its client(s). A client component, desiring for a service to be performed, may send a request to the server via a connector. The server either rejects or performs the request and sends a response back to the client. In this article, we will adopt the following variant: there must be exactly one server within a given architecture and at least one client, and each client must be connected to at least one server. In addition, any other type of component must be contained (possibly indirectly) in either a client or the server.

Our goal is now to specify an *evolution pattern* that evolves the e-shop architecture of Figure 2 into the architecture of Figure 3 that conforms to the Client-Server style. The evolution pattern represents an architectural restructuring that preserves all external ports and structural dependencies of the original architecture. Figure 4 shows the evolution patterns as a UML activity diagram, with two swimlanes representing (a) the automated changes carried out by the *framework*; and (b) the changes manually triggered by the

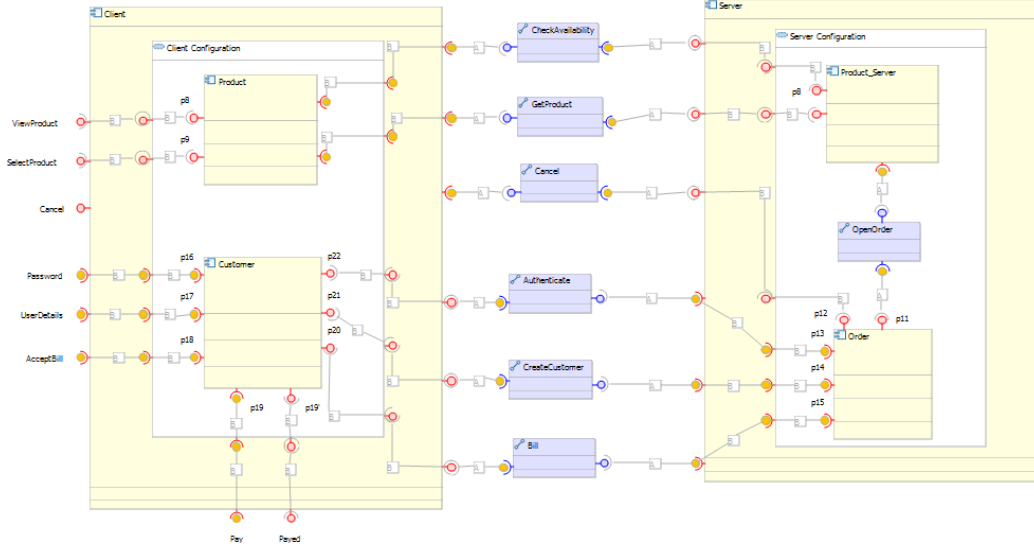


Figure 3: Resulting client-server e-shop architecture in COSA after automatic changes and specific changes triggered by the architect.

architect and executed by the framework. The activity diagram is composed of five basic steps, indicated by dashed rounded rectangles:

1. The *framework* creates one server and n clients ($n \geq 1$) with the names of all new components specified by the architect. For our e-shop architecture, $n = 1$, so one **Server** component and one **Client** component are created.
2. As client(s) and the server are the only top-level components allowed, the architect selects existing components to be moved into a client or into the server. To obtain Figure 3, the architect chooses to move the **Order** component in **Server**, and the **Product** and **Customer** components in **Client**.
3. The *framework* moves the selected components in the server and client(s) and transforms automatically all connectors and dependencies consistently to remain conform to the COSA ADL and the Client-Server architectural style.
4. The architect manually triggers further desired changes. In the example, she wishes to have a **Product** component in both **Server** and **Client** with different ports. It is thus necessary to *split* the **Product** component in two components, **Product** (that allows the user to view

and select products) and **Product_Server** (that encapsulates product information in the server). The architect also wishes to provide the **Cancel** service from the **Server**.

5. The *framework* executes the requested changes and checks whether the architecture still conforms to the Client-Server style and the COSA ADL. In the example, splitting **Product** and moving the resulting component **Product_Server** in **Server** requires to adapt existing dependencies and connectors, yielding the final architecture in Figure 3.

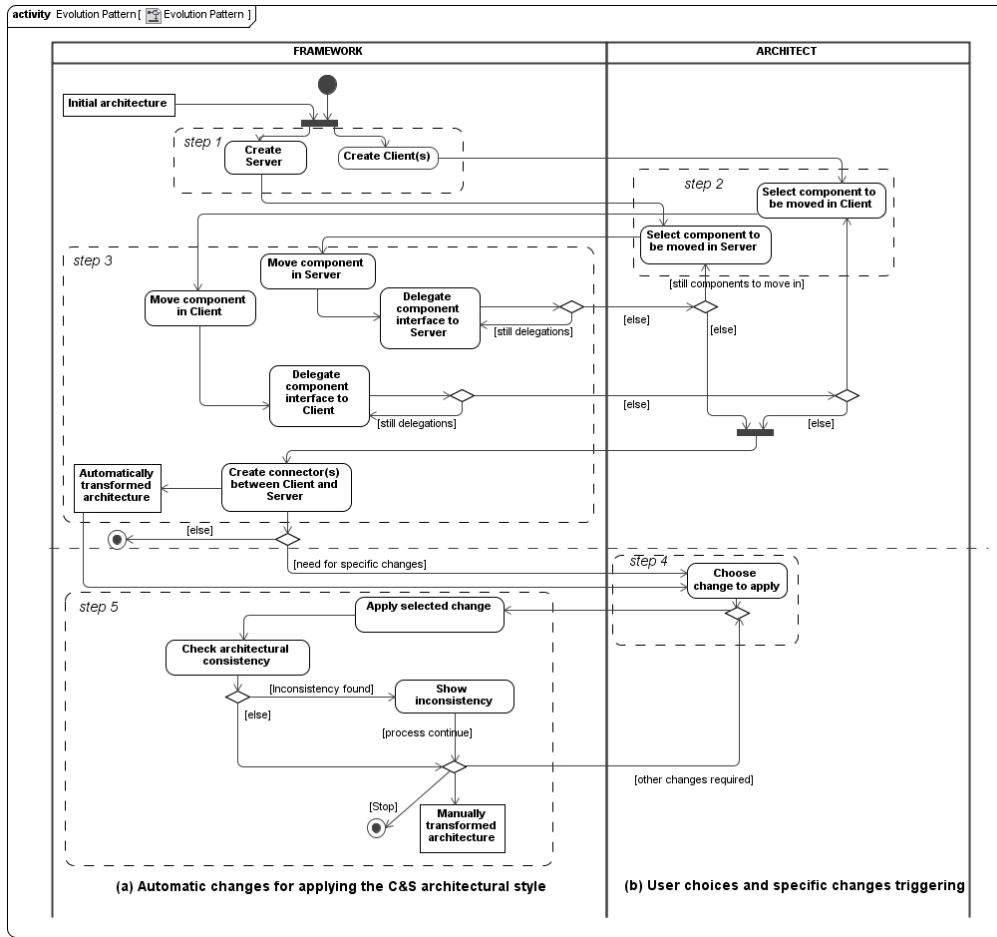


Figure 4: Evolution pattern representing the introduction of the Client-Server architectural style. (In the activities *Delegate component interface to Server* (respectively *Client*) we used the more generic term interface instead of port.)

In section 4 we show how this evolution pattern can be formally specified and analysed, and section 5 shows how it is implemented in the COSABuilder tool.

3.5. Extending COSA with Evolution Operations

The *evolution pattern* of Figure 4 is essentially defined as an application of elementary architectural evolution operations in a particular order. We have identified many such operations in **COSA**, such as: *Move Port* from a component to another, *Split Component* into two or more components, *Merge Components* into one component, *Move in Component* as a subcomponent of another and *Move out Component* from its containing component.

To restructure the e-shop architecture into a Client-Server one, we applied several of these evolution operations. During step 3 of Figure 4, *Move in Component* was used to move components into **Client** or **Server**, followed by transformation *Delegate Component Port* to create necessary bindings and connectors using more primitive transformations like *Move Port* coupled with the creation of corresponding ports and attachments on **Client** and **Server**. During step 5, the *Split component* operation was applied to achieve specific changes requested by the user: splitting the **Product** component (belonging to **Client**) in two components **Product** and **Product.Server**. The latter one is subsequently *moved out* of the **Client** to be *moved in* to the **Server**.

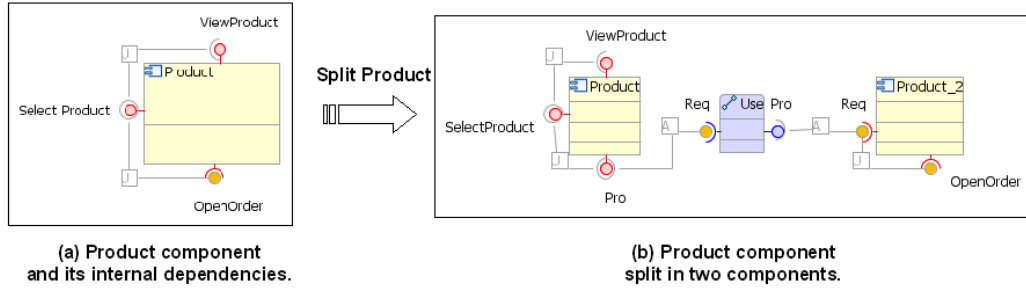


Figure 5: *Split Component* restructuring. (U-rectangles represent *uses* dependencies.)

This interactive evolution operation is guided and restricted by structural constraints, as illustrated in Figure 5. *Split Component* starts by selecting a component (here: *Product*), creating one or more new components (here: *Product_2*¹), and moving some manually selected ports of *Product* into *Prod-*

¹This name is automatically generated but can be changed by the architect.

uct_2. While moving ports, in order to preserve structural dependencies we need to take into account three different situations:

- If the port to be moved does not depend on any other port, it can be moved to the target component without any additional changes.
- If two ports with a *uses*-dependency between them are both moved to the same target component, the internal dependency is moved along.
- If the port to be moved has a *uses*-dependency to or from another port, this dependency needs to be preserved by the evolution pattern. This is for example the case in Figure 5 for *Open Order* port that uses *Select Product*. To preserve the dependency, after moving *OpenOrder* to *Product_2*, a provided port *Pro* on *Product* and a required port *Req* on *Product_2* are created, and a connector is added to connect these two ports. In addition, *uses*-dependencies are respectively created between the new port *Pro* and *SelectProduct* and the new port *Req* and *Open Order*. As a result, the initial *uses* dependency from *Open Order* to *Select Product* is automatically replaced by a longer dependency path between both ports, via an intermediate connector and two newly created ports and two newly created *uses*-dependencies.

4. Formalising and Analysing Architectural Evolution

A prerequisite for providing automatic support for architectural evolution is the ability to formally specify the ADL, architectural styles and evolution patterns. We use graph transformation theory for this purpose. The analogy between architectural evolution and graph transformation is quite natural: an architecture description can be expressed as a graph containing a set of interconnected components. Graph transformations allow us to formally analyse and reason about architectural evolution operations.

In this section we explain how we use AGG² for this purpose, a Java-based graph transformation engine conceived by Taentzer (2004). We exploit AGG's built-in formal analysis mechanisms to reason about evolution patterns and their relation to the architectural styles.

²<http://user.cs.tu-berlin.de/~gragra/agg/>

4.1. Formalising the COSA ADL

Figure 6 shows how the part of the COSA metamodel that is of interest to us is expressed as a *type graph*. Any well-formed architecture can be represented as a graph that conforms to this type graph. The type graph represents all architectural concepts we want to reason about (e.g., Component, Port, Connector, Role) as a *node type*. The associations or relations between the architectural concepts (e.g., bindings, attachments, uses dependencies and containment relationships) are represented by *edge types*.

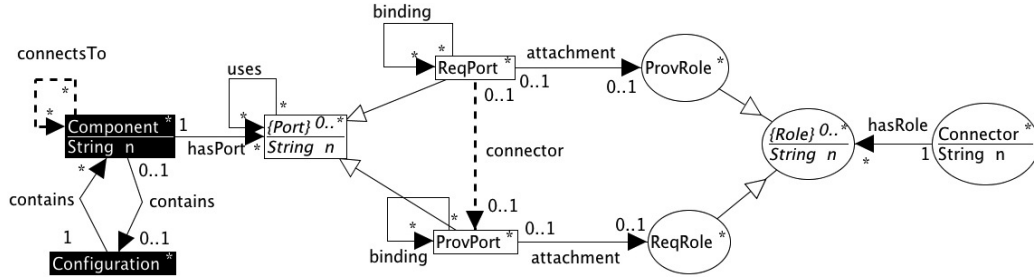


Figure 6: Type graph for a part of the COSA metamodel.

The type graph imposes (lower and upper) multiplicity constraints on edge types. Node and edge types may contain additional attributes, and inheritance can be used between node types, as explained by de Lara et al. (2007). This is the case between the abstract node type *Port* (resp. *Role*) and its two concrete subtypes *ProvPort* (resp. *ProvRole*) and *ReqPort* (resp. *ReqRole*) that represent provided and required ports (resp. roles). All node types contain an attribute *n* of type *String* to represent the name of the corresponding node. A *Port* (resp. *Role*) should always be connected by an edge of type *hasPort* (resp. *hasRole*) to exactly one *Component* (resp. *Connector*), and a *Component* (resp. *Connector*) may have any number of *Ports* (resp. *Roles*). The edge type *contains* relates a component to one of its subcomponents (via an intermediate *Configuration* node type). The *binding* edge type represents the binding of a port of the component to a port (of the same type) belonging to its subcomponent. The *uses* edge type represents a structural internal dependency between ports.

In order to simplify the formal representation of architectures, the type graph also includes so-called “derived” edge types (*connectsTo* and *connector*). Figure 7 shows how they are expressed in terms of a more complex path

of other node types and edge types.

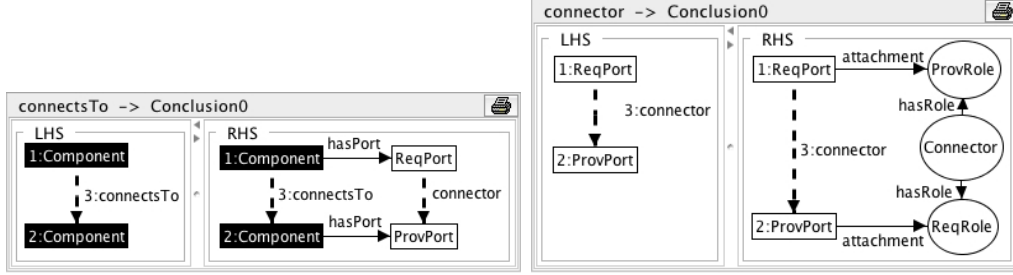


Figure 7: Derived edge types *connector* and *connectsTo* are expressed in terms of a path of other edge types and node types (shown on the right).

In addition to the type graph, graph invariants are needed to specify well-formedness constraints that cannot be expressed directly by the type graph. Some of these are: (i) a component cannot be connected to, or contained in, itself; (ii) a *binding* is only allowed between ports of the same type belonging to a component and one of its subcomponents; (iii) a *uses* dependency is only allowed between different ports belonging to the same component; (iv) two components cannot be at the same time connected to, and contained in, one another. Constraint (ii) and (iii) are formally expressed as graph invariant in Figure 8. The others can be expressed in a similar way.

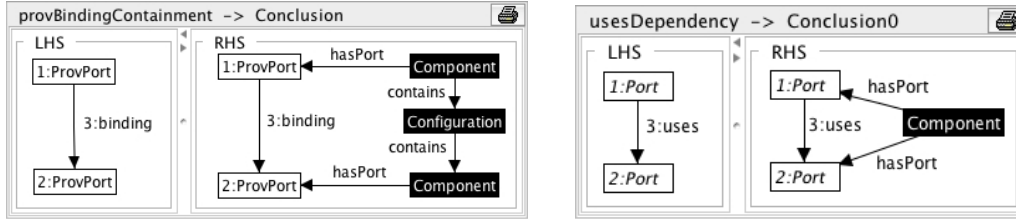


Figure 8: Graph invariants representing additional well-formedness constraints on the type graph.

A COSA architecture can be represented as a *graph* conforming to the type graph together with all of its graph invariants. Figure 9 shows the graph corresponding to the e-shop architecture of Figure 2. This time, we have explicitly shown the *uses*-dependencies that were kept hidden in Figure 2.

With AGG, we can automatically verify that this graph conforms to its type graph and that all imposed graph invariants are satisfied.

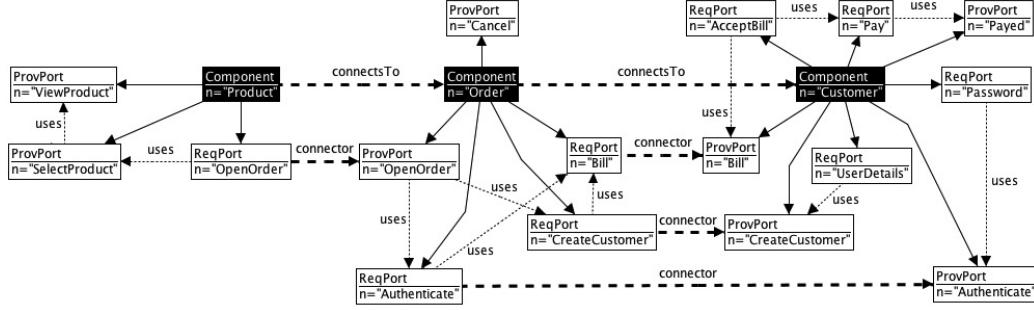


Figure 9: A graph representing the abstract syntax of the initial e-shop architecture.

4.2. Formalising architectural styles

To formalise an architectural style, we proceed in a similar way as for formalising the COSA ADL: we extend the type graph and add graph invariants that express the additional constraints imposed by the architectural style³.

The left part of figure 10 shows how two new *Component* subtypes need to be added to the type graph to represent the Client-Server architectural style : a *Client* node type and a *Server* node type. The node multiplicities state that there should always be one server and at least one client. The edge multiplicities state that each client must be connected to one server.

In addition, we need to add two extra graph invariants that further constrain the type graph: (i) a *Client* component must always be connected to *Server* via one of its ports (see right part of figure 10); (ii) any component that is not a *Client* or *Server* must be contained in another component (i.e., only *Client* and *Server* are allowed as top-level components). For the sake of simplicity, we have not modeled the use of a particular client-server protocol. How to specify and analyse such a protocol is a topic of future work.

A distinct advantage of formalising architectural styles is the ability to verify whether a given architecture is well-formed (i.e., it conforms to its

³Other important information that can typically be found in a style guide Clements et al. (2002), such as the rationale behind the style and the “what it is for and not for” section, have not been formalised in this article. How these aspects of a style can be expressed formally is left as a topic for future work.

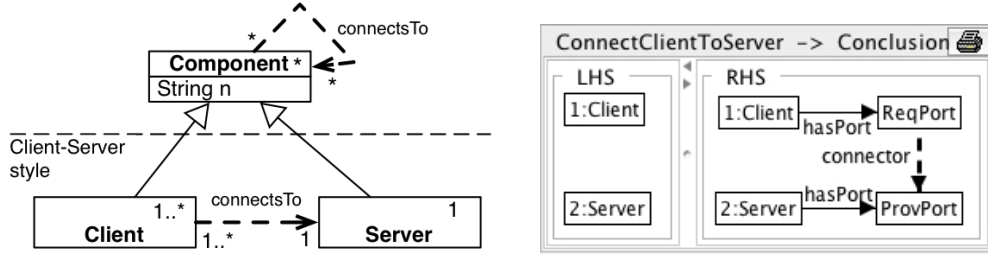


Figure 10: Extension of the COSA type graph to accommodate the Client-Server architectural style. The *connectsTo* association between *Client* and *Server* refines (i.e., constrains) the one on *Component*.

ADL), and whether it conforms to an architectural style. This kind of verification is quite straightforward. We have defined the COSA ADL and the Client-Server architectural style as a type graph together with additional graph constraints, and AGG provides direct support for checking whether a graph conforms to its type graph and its associated graph constraints. This checking was successfully done on our e-shop architecture, before and after introduction of the Client-Server style.

4.3. Formalising evolution pattern

In order to specify architectural evolution patterns we rely on the notion of a *graph transformation*. Essentially, a graph transformation takes a graph as input and produces another graph as output. It is specified by means of a *graph transformation rule* that must conform to the type graph and all graph constraints.

Figure 11 shows three examples of graph transformation rules that formalise activities of Figure 4: *Create Server*, *Move Component To Server* and *Delegate Provided Port to Server*. In general, the specification of a graph transformation rule is composed of three different parts (displayed from left to right in the figure): a number of optional negative application conditions (NAC), a left-hand side (LHS) and a right-hand side (RHS). Applying the transformation proceeds as follows:

1. an occurrence (or “match”) of the LHS needs to be found in the host graph. It is possible that multiple matches are found. In that case, the user selects the desired match or the tool chooses a match non-deterministically.

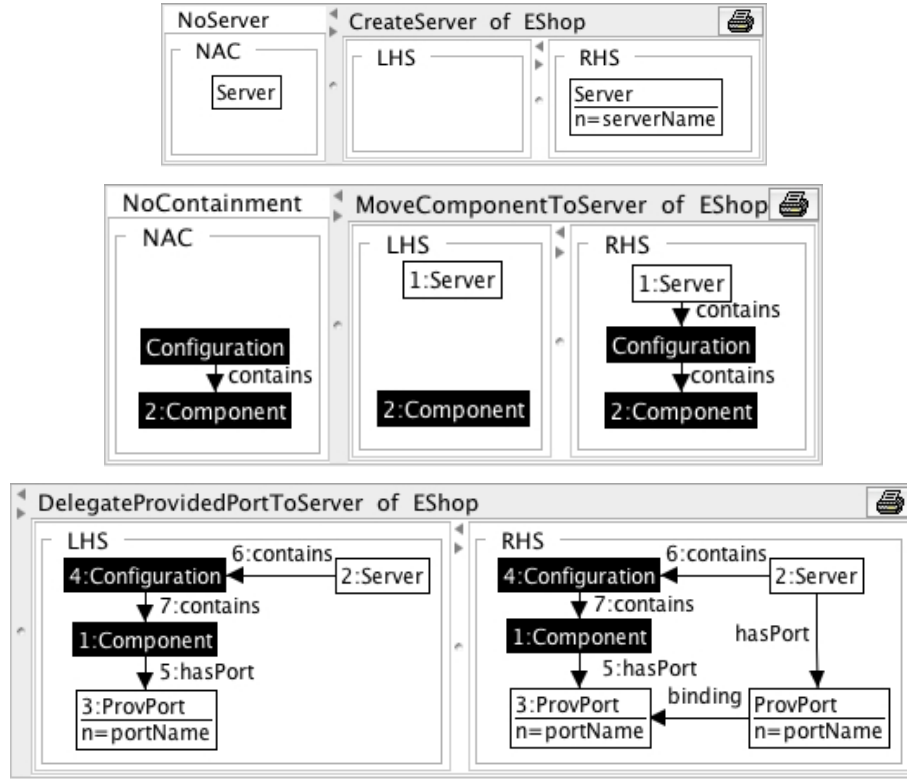


Figure 11: Three graph transformation rules that are part of the evolution pattern to move a component into a server. Next to *DelegateProvidedPortToServer* we defined a similar transformation rule for *DelegateRequiredPortToServer*.

2. if successful, the NAC is used to verify that certain “forbidden constructs” do not appear in the match. In rule *CreateServer* of figure 11, NAC *NoServer* states that a server cannot be created if there is already one⁴. NAC *NoContainment* of rule *MoveComponentToServer* states that the component to be moved is not allowed to be contained in any other configuration.
3. if the NAC is satisfied, the transformation rule is applied by “replacing” the match corresponding to the LHS in the host graph by its RHS. Identical numbers in LHS and RHS are used to identify nodes and edges that are to be preserved by the transformation. Nodes or edges only

⁴In contrast, multiple clients are allowed.

appearing in the RHS are newly added; nodes or edges only appearing in the LHS are removed by the transformation.

Evolution patterns can be formalised, in part, as sequences of graph transformation rules. The rule sequence below specifies the order in which to apply the transformation rules of Figure 11 to move components into the server:

`CreateServer; (MoveComponentToServer)*;
(DelegateProvPortToServer)*; (DelegateReqPortToServer)*`

The `;` symbol specifies the order in which to apply the rules, whereas the `*` is used to specify a repetition of a rule (or even a sequence of rules), by applying it as long as a new match can be found in the graph. Rule sequences provide a way to formalise the *evolution pattern* of Figure 4 that specifies the introduction of the Client-Server architectural style. An activity diagram represents a set of possible execution paths. Each such path can be expressed by a rule sequence. As such, the set of rule sequences represents all possible execution scenarios specified by the evolution pattern.

AGG provides different ways to analyse whether an evolution pattern is well-defined, including *critical pair analysis (CPA)* and *rule sequence analysis*. CPA is used to detect parallel conflicts and sequential dependencies between pairs of transformation rules R_1 and R_2 . *Parallel conflicts* represent situations in which two transformation rules are not jointly applicable to the same host graph: application of rule R_1 prohibits subsequent application of R_2 or vice versa. This is used, for example, to verify whether two rules are mutually exclusive. *Sequential dependencies* represent situations where rule R_2 is causally dependent on R_1 : R_2 cannot be applied directly to the host graph, but becomes applicable once R_1 has been applied.

As an example, consider the evolution pattern of Figure 4. Starting from an initial software architecture, one can execute two parallel sequences of activities⁵. The first sequence starts with *Create Server* followed by *Move Component to Server* and *Delegate (Required or Provided) Port to Server* (step 3). The second sequence is similar but for Client instead of Server.

To verify that the imposed order between the different activities actually makes sense, we specified each of them as a graph transformation rule, and ran the critical pair analysis to detect parallel conflicts and sequential dependencies between pairs of rules. The result of this analysis is shown in

⁵The fork notation in Figure 4, denoted by a black horizontal bar with two outgoing transitions, represents the start of two parallel threads of execution.

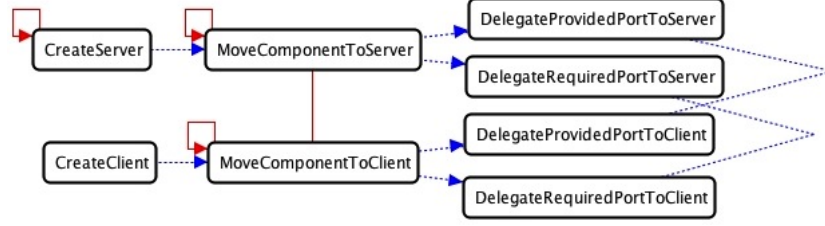


Figure 12: Critical pair analysis of graph transformation rules. Dotted arcs represent sequential dependencies, solid arcs represent parallel conflicts.

Figure 12. It corresponds to our intention: *Create Client* and *Create Server* are parallel independent; they can be applied in parallel without any harm. *Create Server* is in parallel conflict with itself because only one server can be introduced into the client-server architecture. *Move Component to Client* and *Move Component to Server* have a potential parallel conflict if one tries to move the same component in the client and in the server. *Move Component to Server* (resp. *Client*) is in conflict with itself because one cannot move the same component twice. We also find all expected sequential dependencies: *Delegate (Provided or Required) Port to Server* causally depends on *Move Component to Server* that causally depends on *Create Server*. This automated formal analysis makes us more confident that the evolution pattern specified in Figure 4 for applying the Client-Server architectural style is well-defined. In fact, we used the formal analysis as a kind of debugging mechanism: it enabled us to fine-tune the graph transformation rules and the well-formedness constraints imposed by the architectural style.

Rule / Criteria	(1) initialization	(2) no node-deleting rules	(3) no impeding predecessors	(4a) pure enabling predecessor	(4b) direct enabling predecessor(s)
CreateServer					
MoveComponentToServer					CreateServer
DelegateProvInterfaceToServer					MoveComponentToServer
DelegateReqInterfaceToServer					DelegateProvInterfaceToServer

Figure 13: Automated applicability analysis of the rule sequence that specifies how to move components into the server.

As explained by Lambers et al. (2008) and Jurack et al. (2008), it is possible to analyse rule sequences and this analysis is supported by AGG. We can specify any given path in an activity diagram as a rule sequence, verify whether this sequence is applicable and, if not, what are the potential applicability problems. For the example rule sequence defined earlier, Fig-

ure 13 shows the result of analysing applicability (for details, see Lambers et al. (2008)). It reveals that the analysed rule sequence does not have any particular inconsistency (it would have been displayed in red otherwise). As a result, the application of the rules in the specified order will not provoke conflicts. This confirms the results concerning sequential dependencies observed in Figure 12. The rule sequence for moving components into a server thus is well-defined. Analysing the rule sequence to move components into a client yields a similar result.

5. Automating Architectural Evolution

After this formal validation of our ideas, we present now the practical validation through the extension of *COSABuilder*, an Eclipse plug-in that implements the COSA ADL to specify architectural descriptions. Its extension with support for evolution patterns and architectural styles is shown in Figure 14. The right frame displays the initial palette of *COSABuilder* enriched with the concept of *Service Use* to represent uses dependencies between ports. The central frame displays the architecture description. Evolution operations can be applied to it by selecting the appropriate architectural element and choosing the desired evolution operation from a context-sensitive menu. Figure 14 shows the *Move In* operation on the selected *Order* component. Figure 15 shows a new menu that appears (to select the target component into which *Order* needs to be moved), as well as the architecture resulting from applying this evolution operation.

The implemented evolution layer enables the definition of: (i) elementary architectural *evolution operations*, (ii) reified *evolution patterns* in terms of those elementary evolution operations and (iii) *architectural styles* that can be checked. Evolution operations and evolution patterns are defined as first-class entities, by extending the COSA meta-model with two new metaclasses: *EvolutionOperation* and *EvolutionPattern*.

Each evolution operation is implemented as a Java class using the *Singleton* design pattern. It has a unique method *execute(parameters)*. The first parameter is the *context*: the architectural element to be modified. Other parameters are specific to the evolution operation under consideration. We implemented the following elementary evolution operations, in such a way that they preserve internal *uses* dependencies:

- *Create* or *Delete* an architectural element (Component or Connector) or interface (Port or Role).

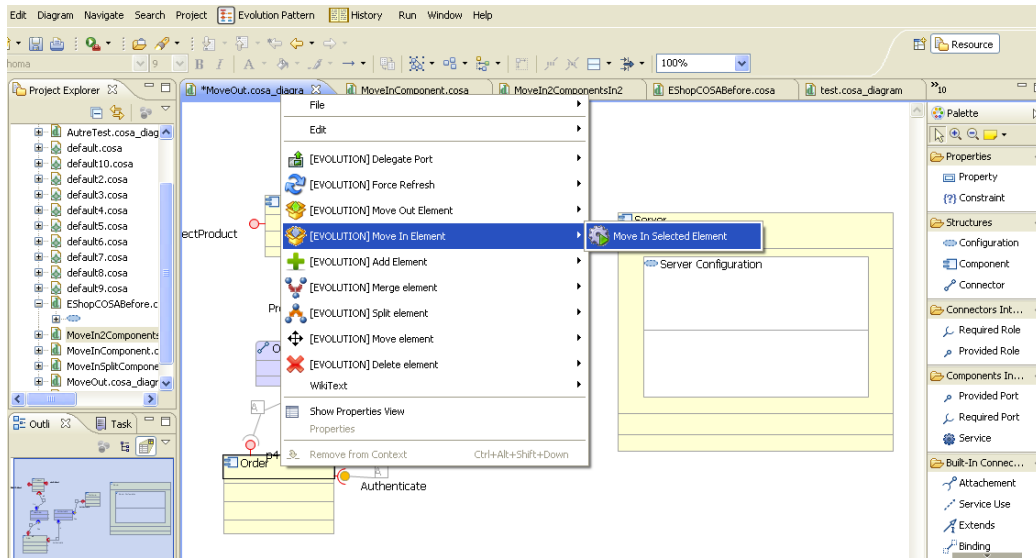


Figure 14: Extension of COSABuilder tool with support for architectural evolution.

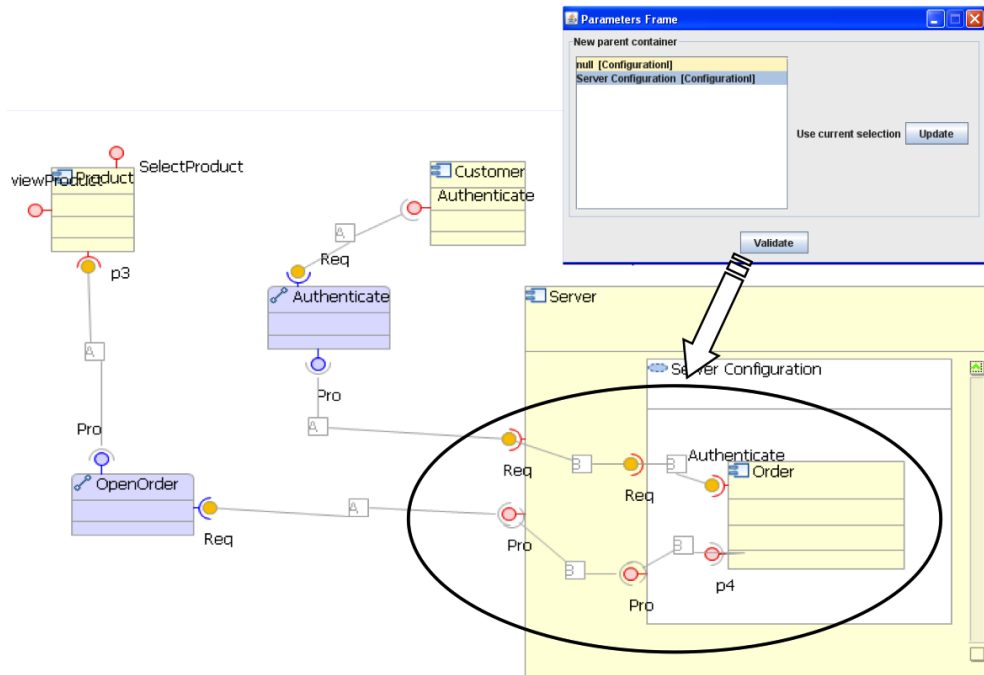


Figure 15: Result of applying the the *Move In* evolution operation in COSAEvol.

- *Move* an architectural element to a new parent.
- *Split* an architectural element into two or more.
- *Merge* several architectural elements into one.
- *Move Out* an element from its containing element, or *Move In* an element inside another one (shown in Figures 14 and 15).
- *Delegate*: create a binding from a given port to another element in the architecture, generally its parent.

Evolution patterns can be implemented in terms of existing evolution operations. Figure 16 illustrates the execution of the evolution pattern of Figure 4 to introduce the Client-Server style.

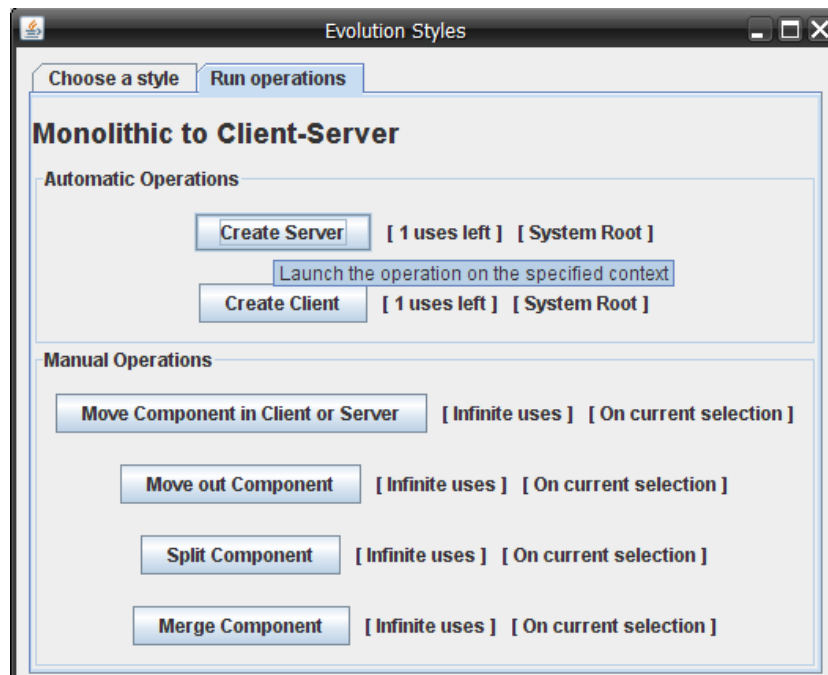


Figure 16: Running implementation of the Client-Server evolution pattern.

Figure 17 illustrates how to check conformance of an architecture to the Client-Server architectural style. This is achieved in three steps: (1) open the Evolution window; (2) select the architectural style to be checked on

the architecture; and (3) display the result of applying the architectural style, together with a message window indicating success or failure of the conformance checking. Figure 18 shows an example of success on the left, and an example of failure on the right.

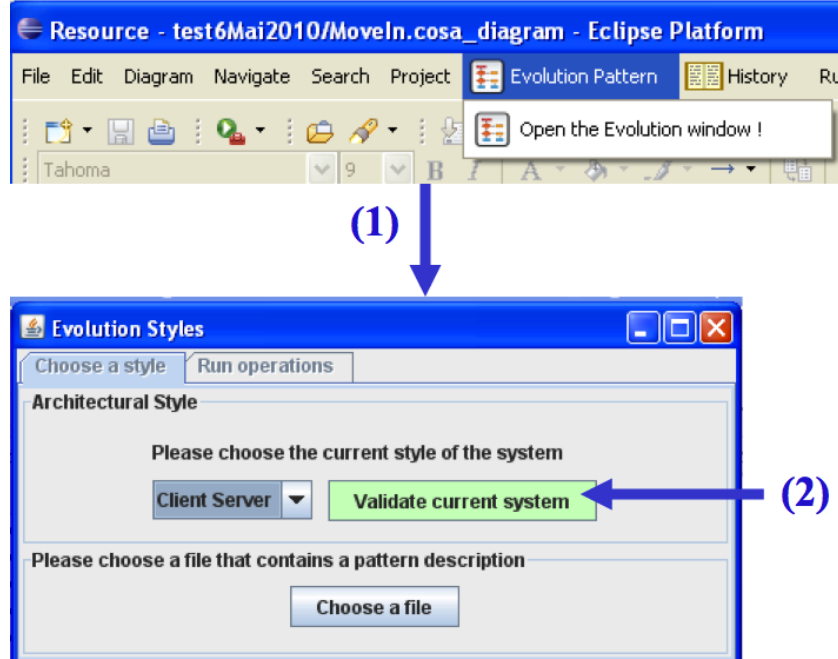


Figure 17: Checking conformance to the Client-Server architectural style.

6. Related Work

A variety of related work focuses on the individual topics that form part of our architectural evolution framework. The novelty of our research is that we combine several topics together by introducing the use of evolution patterns, formally specifying and analysing them using graph transformation to introduce and check architectural styles, and implementing them in an architectural modeling tool for the COSA ADL.

Le Metayer (1998) used graph transformation theory to describe software architecture and architectural style as graph grammars. He proposed to use a coordinator in terms of rules that must be statically checked, as opposed

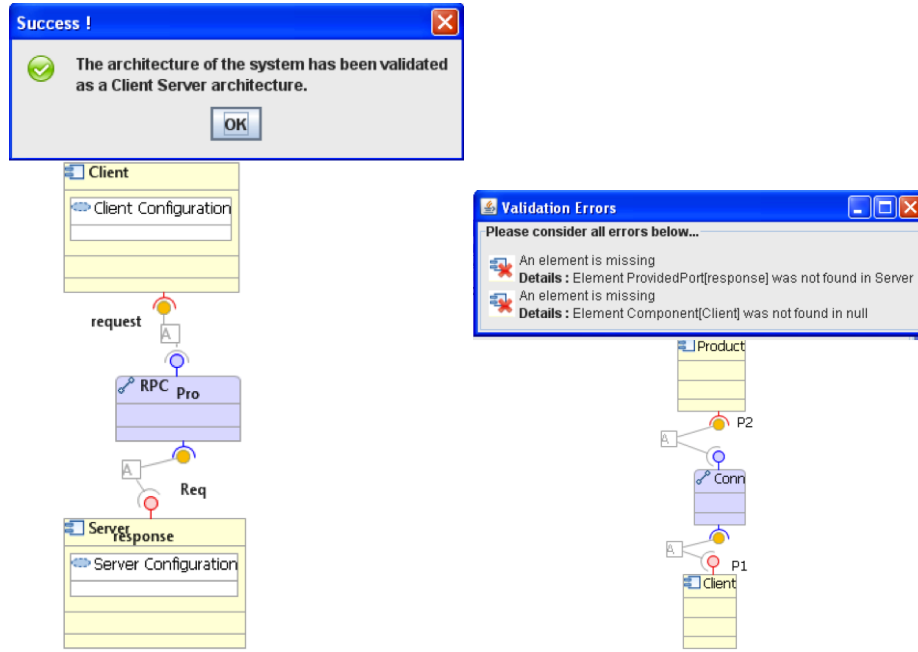


Figure 18: Result of checking conformance to the Client-Server architectural style on two different architectures.

to our use of evolution patterns that introduce architectural styles. Wermelinger and Fiadeiro (2002) used graph transformation theory to present an algebraic foundation for software architecture reconfiguration. Grunske (2005) formalised architectural refactorings as hypergraph transformation rules that can be applied automatically.

The idea of considering architectural styles as typical architectural “evolution patterns” was first introduced by Tamzalit et al. (2006) who proposed the SEM model that introduces the concept of *evolution style*. Garlan (2008) defined an evolution style as a set of evolution paths among different kinds of systems.

McVeigh et al. (2006) provide first-class operations that express and capture architectural changes. The main difference is the type of evolution operations that are provided and the ADL for which they are supported. An additional difference is that we use these evolution operations as elementary building blocks to create more complex evolution patterns to introduce architectural styles.

Noppen and Tamzalit (2010) go beyond the evolution pattern by proposing a framework to tailor evolution processes according to some desired architectural traits by looking for them in a given architectural knowledge base.

7. Future perspectives

The current article only scratched the surface of how one should provide more disciplined support for architectural evolution. This section provides a roadmap of future work that is still required in this very important emerging research domain.

Our formalisation of the structural viewpoint and the Client-Server architectural style, only focused on the architectural elements, relations and structural constraints between them. As explained by Medvidovic and Taylor (2000), many ADLs enable the formal analysis and verification of important non-functional properties such as consistency, completeness, correctness, performance, reliability, security, availability and dependability. We need to integrate support for verifying such properties, and to preserve these properties during architectural evolution. We also need support for documenting the rationale behind an architecture and its imposed architectural styles.

Because **COSA** offers 4 layers of modeling (similar to the OMG MDA architecture), the tool support can accommodate other ADLs as well by considering **COSA** as a pivot ADL. Smeda et al. (2005) proposed strategies to transform an architectural description specified in **COSA** to a description in another ADL such as UML 2. This would allow us to benefit from a wide range of tools that have been implemented for the UML language.

Our case study presented only one viewpoint and architectural style. Applying it to other viewpoints and styles opens up the possibility to evolve an architectural description that involves multiple viewpoints, and to deal with architectural styles that span multiple viewpoints. Other interesting scenarios are the replacement of an architectural style by another one on an existing architecture.

Another future research topic is the study of *co-evolution* between ADLs, architectural styles, their conforming architectures, and their implementation. Changes to any of these entities may require changes to the others. We thus need to analyse this change impact and manage co-evolution while limiting the number of constraint violations. This becomes very challenging when multiple architectural styles co-exist that may interfere when applied to the same software architecture.

8. Conclusion

In this article, we introduced architectural evolution patterns as a disciplined mechanism to introduce architectural styles to an architectural description. We provided a case study using the structural viewpoint and the Client-Server architectural style expressed in the COSA ADL. Using graph transformation, we formally analysed evolution patterns by relying on the notions of critical pair analysis and rule sequence analysis. This allowed us to ensure that the evolution patterns we specified are well-formed and preserve the consistency constraints imposed by the ADL and the architectural style. We implemented and practically validated our ideas by extending COSA-Builder, the tool that accompanies the COSA ADL, with explicit and first class support for defining and applying evolution patterns and architectural styles.

References

- D. E. Perry, A. L. Wolf, Foundations for the Study of Software Architecture, ACM SIGSOFT Softw. Eng. Notes 17 (4) (1992) 40–52.
- M. Shaw, D. Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, ISBN 978-0131829572, 1996a.
- L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, Addison-Wesley Professional, ISBN 0201199300, 1997.
- R. N. Taylor, N. Medvidovic, E. Dashofy, Software Architecture: Foundations, Theory, and Practice, Wiley, ISBN 978-0-470-16774-8, 2009.
- ISO/IEC Standard 42010, IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, ISO/IEC, 2007.
- O. Le Goaer, D. Tamzalit, M. Ouassalah, A. Seriali, Evolution Shelf: Reusing Evolution Expertise within Component-Based Software Architectures, Proc. Int’l Conf. Computer Software and Applications (2008) 311–318.
- D. Garlan, J. M. Barnes, B. Schmerl, O. Celiku, Evolution Styles: Foundations and Tool Support for Software Architecture Evolution, in: Working IEEE/IFIP Conf. Software Architecture and European Conf. Software Architecture, IEEE Press, 131–140, 2009.

- T. Mens, J. Magee, B. Rumpe, Evolving Software Architecture Descriptions of Critical Systems, IEEE Computer .
- P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, R. Little, Documenting software architectures: views and beyond, Pearson Education, 2002.
- P. Kruchten, The 4+1 View Model of Architecture, IEEE Software 12 (6) (1995) 42–50.
- S. Vestal, A cursory Overview and Comparison of Four Architecture Description Languages, Tech. Rep., Honeywell Technology Center, 1993.
- D. Garlan, R. Monroe, D. Wile, ACME: An Architecture Description Interchange Language, in: Proc. CASCON'97, 169–183, 1997.
- D. Garlan, R. Allen, J. Ockerbloom, Exploiting style in architectural design environments, in: Proc. ACM SIGSOFT Symp. Foundations of Software Engineering, ACM, 175–188, 1994.
- N. Medvidovic, D. S. Rosenblum, R. N. Taylor, A language and environment for architecture-based software development and evolution, in: Proc. Int'l Conf. Software Engineering, IEEE Computer Society, ISBN 1-58113-074-0, 44–53, 1999.
- J. Magee, N. Dulay, S. Eisenbach, J. Kramer, Specifying distributed software architectures, in: Proc. European Software Engineering Conf., Springer, 137–153, 1995.
- S. Vestal, P. Binns, Scheduling and communication in MetaH, in: Proc. Symp. Real-Time Systems, 194–200, 1993.
- D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, W. Mann, Specification and analysis of system architecture using Rapide, IEEE Trans. Softw. Eng. 21 (1995) 336–355.
- M. Moriconi, R. Riemenschneider, Introduction to SADL 1.0: A language for specifying software architecture hierarchies, Tech. Rep. SRI-CSL-97-01, SRI International, 1997.

- M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, G. Zelesnik, Abstractions for software architecture and tools to support them, *IEEE Trans. Softw. Eng.* 21 (4) (1995) 314–335.
- R. Allen, D. Garlan, The Wright architectural specification language, Tech. Rep. CMU-CS-96-TBD, Carnegie Mellon University, 1996.
- B. Lewis, Architecture Based Model Driven Software and System Development for Real-Time Embedded Systems, in: *RISSEF*, 249–260, 2002.
- N. Medvidovic, R. N. Taylor, A classification and comparison framework for architecture description languages, *IEEE Trans. Softw. Eng.* 26.
- M. Shaw, D. Garlan, *Software Architecture — Perspectives on an Emerging Discipline*, Prentice Hall, ISBN 0-13-182957-2, 1996b.
- D. Garlan, M. Shaw, An Introduction to Software Architecture, in: V. Ambriola, G. Tortora (Eds.), *Advances in Software Engineering and Knowledge Engineering*, vol. I, World Scientific, 1993.
- H. Gomaa, G. Farrukh, Composition of software architectures from reusable architecture patterns, *Proc. Int’l Software Architecture Workshop (ISAW ’98)* .
- D. Garlan, S.-W. Cheng, A. J. Kompanek, Reconciling the needs of architectural description with object-modeling notations, *Sci. Comput. Program.* 44 (1) (2002) 23–49, ISSN 0167-6423.
- S. Maillard, A. Smeda, M. Oussalah, COSA: An Architectural Description Meta-Model, in: *ICSOFTE (SE)*, 445–448, 2007.
- G. Taentzer, AGG: A Graph Transformation Environment for Modeling and Validation of Software, in: *Proc. AGTIVE 2003*, *Lecture Notes in Computer Science*, Springer, 446–453, 2004.
- J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, G. Taentzer, Attributed graph transformation with node type inheritance, *Theoretical Computer Science* 376 (3) (2007) 139 – 163, ISSN 0304-3975.
- L. Lambers, H. Ehrig, G. Taentzer, Sufficient Criteria for Applicability and Non-Applicability of Rule Sequences, *Electronic Communications of the EASST* 10.

- S. Jurack, L. Lambers, K. Mehner, G. Taentzer, Sufficient Criteria for Consistent Behavior Modeling with Refined Activity Diagrams, in: Proc. Int'l Conf. Model Driven Engineering Languages and Systems (MoDELS'08), Springer, 341–355, 2008.
- D. Le Metayer, Describing software architecture styles using graph grammars, IEEE Trans. Softw. Eng. 24 (7) (1998) 521–533.
- M. Wermelinger, J. L. Fiadeiro, A graph transformation approach to software architecture reconfiguration, Sci. Comput. Program. 44 (2) (2002) 133–155.
- L. Grunske, Formalizing Architectural Refactorings as Graph Transformation Systems, in: Int'l Conf. SNPD and ACIS Int'l Workshop SAWN, 324–329, 2005.
- D. Tamzalit, M. Oussalah, O. Le Goaer, A. Seriai, Updating Software Architectures: a style-based approach, in: Proc. Int'l Conf. Software Engineering Research and Practice, CSREA, ISBN ISBN 1-932415-90-4, 336–342, 2006.
- D. Garlan, Evolution Styles Formal Foundations and Tool Support for Software Architecture Evolution, Tech. report. CMU-CS-08-142, CMU .
- A. McVeigh, J. Kramer, J. Magee, Using resemblance to support component reuse and evolution, in: Proc. Conf. Specification and Verification of Component-based Systems, ACM, ISBN 1-59593-586-X, 49–56, 2006.
- J. Noppen, D. Tamzalit, ETAK: Tailoring Architectural Evolution by (re-)using Architectural Knowledge, 5th Workshop on SHaring and Reusing architectural Knowledge (SHARK 2010), ICSE 2010. .
- A. Smeda, M. Oussalah, T. Khammaci, MADL: Meta Architecture Description Language, in: SERA, 152–159, 2005.